Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 379 (2007) 418–436
Theoretical Computer Science
www.elsevier.com/locate/tcs

# Efficient pebbling for list traversal synopses with application to program rollback☆

Yossi Matias[a,*], Ely Porat[b,a]

[a] School of Computer Science, Tel Aviv University and Google Inc, Tel Aviv, Israel
[b] Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel

**Abstract**

We show how to support efficient back traversal in a unidirectional list, using small memory and with essentially no slowdown in forward steps. Using $O(\lg n)$ memory for a list of size $n$, the $i$'th back-step from the farthest point reached so far takes $O(\lg i)$ time in the worst case, while the overhead per forward step is at most $\epsilon$ for arbitrary small constant $\epsilon > 0$. An arbitrary sequence of forward and back steps is allowed. A full trade-off between memory usage and time per back-step is presented: $k$ vs. $kn^{1/k}$ and vice versa. Our algorithms are based on a novel pebbling technique which moves pebbles on a virtual binary, or $n^{1/k}$-ary, tree that can only be traversed in a pre-order fashion.

The compact data structures used by the pebbling algorithms, called list traversal synopses, extend to general directed graphs, and have other interesting applications, including memory efficient hash-chain implementation. Perhaps the most surprising application is in showing that for any program, arbitrary rollback steps can be efficiently supported with small overhead in memory, and marginal overhead in its ordinary execution. More concretely: let $P$ be a program that runs for at most $T$ steps, using memory of size $M$. Then, at the cost of recording the input used by the program, and increasing the memory by a factor of $O(\lg T)$ to $O(M \lg T)$, the program $P$ can be extended to support an arbitrary sequence of forward execution and rollback steps: the $i$'th rollback step takes $O(\lg i)$ time in the worst case, while forward steps take $O(1)$ time in the worst case, and $1 + \epsilon$ amortized time per step.


## 1. Introduction

A unidirectional list enables easy forward traversal in constant time per step. However, getting from a certain object to its preceding object cannot be done efficiently. It requires forward traversal from the beginning of the list and takes time proportional to the distance to the current object, using $O(1)$ additional memory. In order to support more effective back steps on a unidirectional list, it is necessary to add auxiliary data structures.

☆ A preliminary version of this paper was presented at [Y. Matias, E. Porat, Efficient pebbling for list traversal synopses, in: International Colloquium on Automata, Languages and Programming, 2003, pp. 918–928].

* Corresponding author.
*E-mail addresses:* matias@cs.tau.ac.il, matias@cs.stanford.edu (Y. Matias), porately@cs.biu.ac.il (E. Porat).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.02.048

Trailing pointers, namely backward pointers from the current position to the beginning of the list, can be easily maintained in $O(1)$ time per forward step, and support back steps in $O(1)$ time. However, the memory required for maintaining trailing pointers is $\Theta(n)$, where $n$ is the distance from the beginning of the list to the farthest point reached so far. A simple time-memory trade-off can be obtained by keeping a pointer every $n/k$ forward steps. With memory of size $\Theta(k)$, each back step can be done in $\Theta(n/k)$ time. This provides a full generalization of the two previous solutions, with $\Theta(n)$ memory-time product.

A substantially better trade-off can be obtained, using what we call *skeleton* data structures. These skeletons enable full back traversals in $O(kn^{1/k})$ amortized time per back-step, using $k$ additional pointers [1]. However, if one wishes to support fully dynamic list traversal consisting of an arbitrary sequence of forward and back steps, then managing the pointers positions becomes challenging. For the further restriction that forward steps do not incur more than constant overhead (independent of $k$), the problem becomes even more difficult.

The goal of this work is to support memory- and time-efficient back traversal in unidirectional lists, without essentially increasing the time per forward traversal. In particular, under the constraint that forward steps should remain constant time, we would like to minimize the number of pointers kept for the list, the memory used by the algorithm, and the time per back step, while supporting an arbitrary sequence of forward and back steps. Such algorithms can be thought of as *pebbling algorithms*, which maintain pebbles on the list nodes. A pebble may represent a pointer to a node.

We assume that the unidirectional list is already given, and we have access to the list but no control over its implementation. The list may represent a data structure implemented in computer memory or in a database, or it may reside on a separate computer system. The list may also be a virtual one. For instance, it can represent a computational process, where each list node is a configuration state in the computation and the *next* pointer represents a computational step. Supporting efficient back traversal on the list enables effective program rollback, and requiring $O(1)$ time per forward step implies that forward execution of the program is not significantly affected. If the list is a virtual one, then a pebble represents the *content* of the pebbled list node.

### 1.1. Contributions

The main result of this paper is an algorithm that supports efficient back traversal in a unidirectional list, using small memory and with essentially no slowdown in forward steps: $1+\epsilon$ amortized time per forward step for arbitrary small constant $\epsilon > 0$, and $O(1)$ time in the worst case. Using $O(\lg n)$ memory, back traversals can be supported in $O(\lg n)$ time per back step, where $n$ is the distance from the beginning of the list to farthest point reached so far. This result optimal, due to a lower bound of Coppersmith and Jakobsson [5]. In fact, we show that a back traversal of limited scope can be executed more effectively: $O(\lg i)$ time for the $i$'th back step from the farthest point reached so far, for any $i \leq n$, using $O(\lg n)$ memory.

More generally, the following trade-offs are obtained: $O(kn^{1/k})$ time per back step, using $k$ additional pointers, or $O(k)$ time per back step, using $O(kn^{1/k})$ additional pointers; in both cases supporting $O(1)$ time per forward step (independent of $k$). Our results extend to backtracking on general directed graphs, with additional memory of $\lg d_v$ bits for each node $v$ along the backtrack path, where $d_v$ is the outdegree of node $v$.

The crux of the list traversal algorithm is an efficient pebbling technique which moves pebbles on virtual binary or $t$-ary trees that can only be traversed in a pre-order fashion. We introduce the *virtual pre-order tree* data structure which enables managing the pebbles positions in a concise and simple manner.

At every point of time, the configuration of pebbles can be considered as a synopsis of the list traversal done so far. The purpose of this synopsis is to allow effective back traversal. The set of pebbles and the data structures used for their maintenance, combined, are hence denoted as *list traversal synopsis*. This notion is quite different than the notion of *data synopses* [8] that typically serve to represent data sets for particular classes of queries.

The pebbling algorithms involve the maintenance of the list traversal synopsis; they involve both *list operations* and *synopsis maintenance operations*. A list operation includes moving or copying pebbles on the list, and is denoted as a *list step*. Note that list steps could be quite costly. For actual lists (such as data structures) a list-step translates to a forward traversal step which may be expensive if, say, the list is on a remote computer. For virtual lists (such as in computational processes) a list step translates to an update of the configuration state that could be expensive if, say, the state is very large.

### 1.2. Applications

Consider a program $P$ running in time $T$. Then, using our list pebbling algorithm, the program can be extended to a program $P'$ that supports rollback steps, where a rollback after step $i$ means that the program returns to the configuration it had after step $i-1$. Arbitrary ad-hoc rollback steps can be added to the execution of the program $P'$ at a cost of increasing the memory requirement by a factor of $O(\lg T)$, and having the $i$'th rollback step supported in $O(\lg i)$ time. The overhead for the forward execution of the program can be kept an arbitrary small constant. This result is obtained by having a program represented by a linked list in which every node represents a sequence of program states.

Allowing effective rollback steps may have interesting applications. For instance, a desired functionality for debuggers is to allow pause and rollback during execution. Another implication is the ability to take simulation programs and allow running them backward in arbitrary positions. Thus a program can be run with $\epsilon$ overhead in its normal execution, while allowing pausing at arbitrary points, and running it backward an arbitrary number of steps with logarithmic time overhead per back step. The memory requirement is equivalent to state configuration of $\lg T$ points, and additional $O(\lg T)$ memory. Often, debuggers and related applications avoid keeping full program states by keeping only differences between the program states ("delta-encoding"). We obtain our program rollback result by having this approach combined with the list traversal synopsis technique.

Our pebbling technique can be used to support backward computation of a hash chain in time $O(kn^{1/k})$ using $k$ hash values, or in time $O(k)$ using $O(kn^{1/k})$ hash values, for any $1 \leq k \leq \lg n$. A *hash-chain* is obtained by repeatedly applying a one-way cryptographic hash function, starting with a secret seed. There are a number cryptographic applications to hash chains, including password authentication [14], micro-payments [19], forward-secure signatures [10,13], and broadcast authentication protocol [17]. Our results enable effective implementation with arbitrary memory size.

The list pebbling algorithm extends to directed trees and general directed graphs. Applications include the effective implementation of the parent function ("..") for XML trees, and effective graph traversals with applications to "light-weight" Web crawling and garbage collection.

### 1.3. Related work

If it is allowed to change pointers in the list, then one can support arbitrary back traversal using the Schorr–Waite algorithm [20]. This algorithm enables constant time back-step by simply utilizing the fields of that the "next" pointers at the nodes from the head of the list to the current position, to hold pointers to the previous nodes. Constant size auxiliary memory is sufficient to support this "in place" algorithm. The Schorr–Waite algorithm also extends for trees, dags, and general directed graphs.

Another solution for an in-place encoding which supports back traversal, due to Siklossy [22], is based on the following technique. For each node $v$, instead of keeping the pointer next($v$), we keep enc($v$) = prev($v$) XOR next($v$). As in the Schorr–Waite algorithm, only two auxiliary pointers are required — for the current position $u$ and for one of the adjacent positions $u'$, since the other adjacent position $u''$ can be computed as $u'' = \text{enc}(v)$ XOR $u'$. At any position $v$, moving forward and backward can be done in constant time using the encoded information. This algorithm has the advantage that the list encoding remains intact during traversal, and unlike for the Schorr–Waite algorithm, multiple users can traverse the list.

Recall that both the Schorr–Waite and the XOR-based algorithms do not fit the requirement that the list cannot be altered. In particular, these algorithms cannot be used for the applications in which the list represents a computation.

The Schorr–Waite algorithm [20] has numerous applications; see, e.g. [23,25,4]. It would be interesting to explore to what extent these applications could benefit from the non-intrusive nature of our algorithm. There is an extensive literature on graph traversal with bounded memory; see, e.g. [9,2]. Pebbling models were extensively used for bounded space upper and lower bounds. See e.g. the seminal paper by Pippenger [18] and more recent papers such as [2]. The above papers are concerned with different problems from the one addressed here.

The closest work to ours are the papers of Chandra [3] and the paper of Ben-Amram and Petersen [1]. They present a clever algorithm that, using memory of size $k \leq \lg n$, supports back step in $O(kn^{1/k})$ time. However, in their algorithm forward steps take $O(k)$ time. Thus, their algorithm supports $O(\lg n)$ time per back step, using $O(\lg n)$ memory but with $O(\lg n)$ time per forward step, which is unsatisfactory in our context. Ben-Amram and Petersen also

prove a near-matching lower bound, implying that to support back traversal in $O(n^{1/k})$ time per back step it is required to have $\Omega(k)$ pebbles. Our algorithm supports similar trade-off for back steps as the Ben-Amram Petersen algorithm, while supporting simultaneously constant time per forward step. In addition, our algorithm extends to support $O(k)$ time per back step, using memory of size $O(kn^{1/k})$, for every $k \leq \lg n$. Thus, a full range time-memory tradeoff is obtained. Also, the cost of a back step in our algorithm is sensitive to the distance $i$ from the farthest point reached so far – $O(\lg i)$ instead of $O(\lg n)$ – which could be quite significant for the program rollback application.

Recently, and independently to our work, Jakobsson and Coppersmith [11,5] proposed a so-called fractal-hashing technique that enables backtracking hash chains in $O(\lg n)$ amortized time using $O(\lg n)$ memory. Thus, by keeping $O(\lg n)$ hash values along the hash chain, their algorithm enables, starting at the end of the chain, to get repeatedly the preceding hash value in $O(\lg n)$ amortized time. They also provide a lower bound, showing that any pebbling algorithm using $k$ pebbles requires $\frac{1}{4k}\lg^2 n$ time. Thus, for using $k = \lg n$ pebbles, this provides an $\Omega(\lg n)$ bounds on the number of required list steps per back step. Subsequently, Sella [21] showed how to generalize the fractal hashing scheme, to support back steps in $O(k)$ time, for any $k < \lg n$, by holding $O(kn^{1/k})$ hash values. More recent works apply some of these ideas to merkle trees [12,24]. These works are only in the context of hash chains and do not deal with efficient forward traversal. In contrast, our work is concerned with supporting constant time per forward step, while still supporting the most effective back traversal using the smallest possible memory. Note also that our pebbling algorithm enables a full memory-time trade-off for hash-chain execution; that is, both $O(k)$ memory and $O(kn^{1/k})$ time per back step and vice versa. In addition, it is guaranteed that the time per back step is bounded in the worst case.

The most challenging aspect of our algorithm is the proper management of the pointers positions under the restriction that forward steps have very little effect on their movement, to obtain $O(1)$ time per forward step. This is crucial especially for the program rollback application, in which the $\epsilon$-overhead per forward step that we obtain is even more desirable. This is obtained by using the virtual pre-order tree data structure and other techniques, to manage the positions of the back-pointers in a concise and simple manner.

### *1.4. Outline*

The rest of the paper is organized as follows. In Section 2 we describe a model for our list pebbling algorithm; we also present a so-called skeleton data structure that provides some intuition about the pebbling techniques, and the virtual pre-order tree data structure, which will be used by all our algorithms. In Section 3 we describe the list pebbling algorithm, which obtains $O(\lg n)$ amortized time per back step using $O(\lg n)$ pebbles, while supporting $O(1)$ time per forward step. The full list-pebbling algorithm is described in Section 4. It supports $O(\lg n)$ time per back-step in the worst case, using $\lg n$ pebbles, as well as $\epsilon$ overhead per forward step. An extension of the algorithm to support full time-memory trade-off of $O(k)$ vs. $O(kn^{1/k})$ is described in Section 5. In Section 6 we describe the application for efficient reversal of program execution, and for efficient processing of hash chains. Extensions to trees and other graphs are given in Section 7, and we conclude in Section 8. Earlier versions of this paper appear in [15,16].

## 2. Basics

In this section we describe a model for list pebbling, illustrate the basic idea of the list pebbling algorithm, and demonstrate it through a limited functionality of having a sequence of back steps only.

We first describe in Section 2.1 a model that allows to account for list operations versus other synopsis maintenance operations. The model, named PSP, is used in particular for proper accounting in the program rollback application.

We then describe in Section 2.2 algorithms based on *skeleton* data structures, of which the most advanced supports a sequence of back steps in $O(\lg n)$ amortized time per back step, using $\lg n$ pebbles. These data structures are similar in nature to the ones used by [1,11,5,6].

We present in Section 2.3 the *virtual pre-order tree* data structure, which will enable control over the pointers positioning in our pebbling algorithms, and show how it can support the sequence of back steps similarly to the skeleton data structure.

### *2.1. The Pebble Service Provider (PSP) model*

To account separately for the list operations and the synopsis maintenance operations, as well as to address the variety of applications which may be represented by a linked list, we assume that the given list could be accessed via

a third party, denoted as the *PSP*, for *pebble service provider*. The PSP holds pebbles, where each pebble represents a list node whose actual representation is application dependent. In the singly-linked list data structure application, a pebble represents a pointer to a list node; in the computational process application, a pebble may represent the description of a program state, or alternative encoding as will be described later.

The pebbles are represented by array entries, and the array index of a pebble is the *pebble identifier*. A pebbling algorithm communicates to the PSP only instructions of type *create, free, advance, next*, and *fetch,* with pebble identifiers. Initially, the PSP holds a single pebble, representing the first node of the list. The *create(j)* instruction adds a new pebble to the PSP with identifier $j$, which we will refer to also as pebble $j$. The *free(j)* instruction removes pebble $j$ from the PSP; the *advance(j)* instruction updates pebble $j$ that represents a node $v$ in the list to represent the node $u$ that is the successor to $v$ in the list; the *next(j, j′)* instruction updates pebble $j$ to represent the node $u$, which is the successor of the node $v$ represented by pebble $j'$. Finally, the *fetch(j)* instruction returns the content of the node represented by pebble $j$. Both *advance* and *next* instructions are referred to as *list-step* operations. Note that the *advance* instruction is identical to a regular forward step on the given list. The *next* instruction may involve a full duplication of a pebble content, which for some applications could be substantially more expensive than the *advance* instruction, and our objective is therefore to minimize its usage.

For the PSP model, our main result is a *list pebbling algorithm* that uses $\lg n$ pebbles and $O(\lg n)$ memory, to support the $i$'th back step from the farthest point reached so far in $O(\lg i)$ list-steps and $O(\lg i)$ time, and each forward step in $O(1)$ time and no additional list steps.

When using the list pebbling algorithm, for each forward step the *next* instruction is used instead of the original *advance* instruction, used for an ordinary list traversal. The fraction of times in which such replacement occurs can be reduced to $\epsilon$, for arbitrary small $\epsilon > 0$, while also reducing the amortized time overhead per forward step to $O(\epsilon)$, at the cost of increasing the cost of back steps by an *additive* overhead of at most $1/\epsilon$ list steps of type *advance*, and $O(1/\epsilon)$ time.

### 2.2. The skeleton data structure

In this subsection we illustrate simple skeleton data structures and demonstrate them through a limited functionality of having a sequence of back steps only. Note that a full algorithm must support an arbitrary sequence of forward and backward steps, and we will also be interested in refinements, such as reducing to minimum the number of pebbles. Adapting the skeleton data structures to support the full algorithm and its refinements may be quite complicated, since controlling and handling the positions of the various pointers becomes a challenge. For the further restriction that forward steps do not incur more than constant overhead (independent of $k$), the problem becomes even more difficult and we are not aware of any previously known technique to handle this.

As a motivating example, we outline first how to obtain $O(\sqrt{n})$ amortized time per back step, using two additional pointers, $p$ and $p'$. For simplicity, let us only describe how to implement a sequence of back-traversals from position $n$ to the beginning of the list. When the current position is node $n$, pointer $p'$ acts as a *shadow pointer*, and points to position $n - \sqrt{n}$. As long as the current position is between pointer $p'$ and position $n$, a back step is implemented by advancing the pointer $p$, which acts as an *assisting pointer*, from position $p'$, until next($p$) becomes the current position. The condition is detected either using a counter, or by testing equality between the pebbles. When the current position becomes $p'$, we will reposition $p'$ $\sqrt{n}$ positions backwards to $n - 2\sqrt{n}$ by moving the assisting pointer $p$ forward $n - 2\sqrt{n}$ steps starting from the beginning of the list. An update in the position of of $p'$ occurs only after $\sqrt{n}$ back steps are executed since the previous update. Therefore, the amortized cost per back step is $O(\sqrt{n})$. It is straightforward to extend this into a back traversal all the way to the beginning of the list.

An improved, $O(n^{1/3})$ amortized time per back step can be obtained by having the shadow pointer $p'$ positioned at location $n - n^{2/3}$, and adding a second shadow pointer $p''$, pointing initially to position $n - n^{1/3}$. As long as the current position is between pointer $p''$ and position $n$, a back step is implemented by advancing the assisting pointer $p$, from position $p''$, $O(n^{1/3})$ steps until next($p$) becomes the current position. When the current position becomes $p''$, we will update $p''$ to be $n - 2n^{1/3}$ by moving forward $n^{2/3} - 2n^{1/3}$ steps starting from position $p'$; this will occur after $n^{1/3}$ steps each taking $O(n^{1/3})$ time. When the position of $p''$ becomes $p'$ (after $n^{2/3}$ steps each taking $O(n^{1/3})$ time), we will update $p'$ to be $n - 2n^{2/3}$, by moving forward $n - 2n^{2/3}$ steps starting from the beginning of the list. This results with at most $3n^{1/3}$ amortized time per back step.

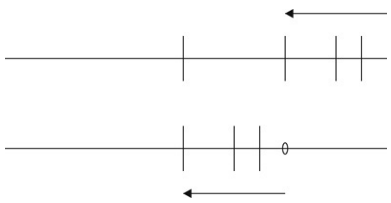

Fig. 1. The skeleton data structure.

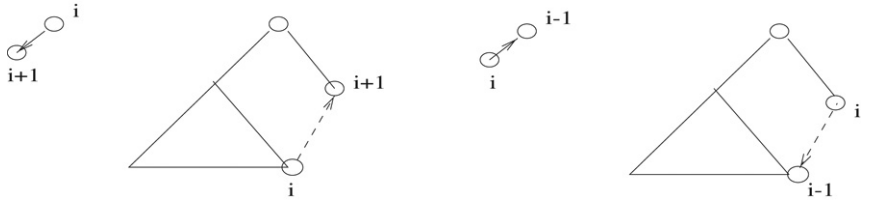


Fig. 2. Preorder traversal: forward steps (left) and backward steps (right).

Using a related technique, a full back-traversal can be implemented, in which each back step takes $O(1)$ amortized time, using $O(\sqrt{n})$ additional pointers. When positioned at node $n$, we keep $\sqrt{n}$ shadow pointers at positions $n-\sqrt{n}+1$ through $n$, as well as $\sqrt{n}$ pointers at positions $n-i\sqrt{n}$, for $i=1,\ldots,\sqrt{n}-1$. As long as the current position is between $n-\sqrt{n}+1$ and $n$, each back step takes $O(1)$ time. When the current position reaches position $n-\sqrt{n}$, the $\sqrt{n}$ shadow pointers are moved to positions $n-2\sqrt{n}+1$ through $n-\sqrt{n}$, in $O(\sqrt{n})$ time, or $O(1)$ amortized time per back-step.

These methods can be extended to more generally support full back traversals in $O(kn^{1/k})$ amortized time per back step, using $k$ additional pointers, or in $O(k)$ amortized time per back step, using $O(kn^{1/k})$ additional pointers.

The skeleton data structure, described next, supports a sequence of back steps only in $O(\lg n)$ amortized time per back step, using $\lg n$ pebbles.

Let $n$ be the current position and assume that $n$ is a power of 2. We maintain $\lg n+1$ pebbles between the current position and the beginning of the list, where the $i$'th pebble is at distance $2^i$ from the current position, $i=0,1,\ldots,\lg n$. Denote this as the *skeleton* data structure of size $n$.

A sequence of back traversals from position $n$ to the position of the $i$'th pebble is done as follows:

1. Have a sequence of back traversals from position $n$ to the position of the $i-1$st pebble, using the skeleton data structure. All pebbles between these positions are freed.
2. Build a skeleton data structure of size $2^{i-1}$ between the positions of the $i$'th and $i-1$st pebbles, using the pebbles freed in Step 1. (See Fig. 1.)
3. Have a sequence of back traversals from the position of the $i-1$st pebble to the position of the $i$'th pebble, using the skeleton data structure between these points.

Steps 1 and 3 are recursive applications of the algorithm for problems of size $2^{i-1}$. Step 2 is implemented in a single sequence of $2^{i-1}$ forward steps. The time required to have a sequence of back steps from position $n$ to the position of the $i$'th pebble is therefore $T(i)=2T(i-1)+2^{i-1}$, $T(1)=O(1)$, implying $T(i)=O(i2^i)$, and an amortized $O(\lg d)$ time for the $d$'th back-step.

### 2.3. The virtual pre-order tree data structure

The reader is reminded (see Fig. 2) that in a pre-order traversal, the successor of an internal node in the tree is always its left child; the successor of a leaf that is a left child is its right sibling; and the successor of a leaf that is a right child is defined as the right sibling of the nearest ancestor that is a left child. An alternative description is as follows: consider the largest subtree of which this leaf is the rightmost leaf, and let $u$ be the root of that subtree. Then the successor is the right sibling of $u$. Consequently, the: *backward traversal* on the tree will be defined as follows. The successor of a node that is a left child is its parent. The successor of a node $v$ that is a right child is the rightmost leaf of the left subtree of $v$'s parent.

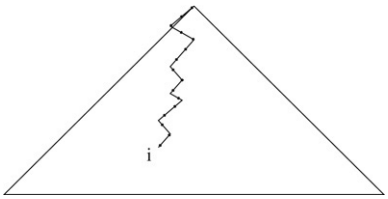


Fig. 3. A pebbled path from the root to current position $i$.

The *virtual pre-order tree* data structure consists of (1) an implicit binary tree, whose nodes correspond to the nodes of the linked list, in a pre-order fashion, and (2) an explicit subtree of the implicit tree, whose nodes are pebbled. For the basic algorithm, the pebbled subtree consists of the path from the root to the current position.

Each pebble represents a pointer; i.e. pebbled nodes can be accessed in constant time. We defer to later sections the issues of how to maintain the pebbles, and how to navigate within the implicit tree, without actually keeping it. Starting at node $n$, a back-traversal can be executed while maintaining $\lg n$ pebbles with $O(\lg n)$ amortized time per back-step, as follows (see Fig. 3):

If node $i$ is a left child, then node $i-1$ is the parent of $i$, and the path from the root to node $i-1$ is already pebbled. Therefore, doing the backtrack step as well as updating the data structure are trivial.

If node $i$ is a right child, then node $i-1$ is the rightmost leaf in the sub-tree, $T'$, whose root is the left sibling of node $i$. In this case the path from the root of $T'$ to node $i-1$ (consisting of going down $T'$ through right children only), is yet to be pebbled. The challenge is that getting into these nodes requires a full traversal of $T'$.

Let $T$ be the sub-tree whose root is node $i$, and let $t$ be the size of $T$. Note that the size of $T'$ is also $t$. Thus, moving from node $i$ to node $i-1$, as well as pebbling the path from the root of $T'$ to node $i-1$ takes $t$ steps. We will charge this cost to the sequence of all backtrack steps within $T$, i.e. starting from the rightmost leaf in $T$ and getting to its root (node $i$).

The total cost of all backtrack steps is $C = \sum_v t(v)$, where $v$ is a right child and $t(v)$ is the size of the subtree rooted at $v$. It is easy to verify that $C < \frac{n \lg n}{2}$, resulting with amortized $O(\lg n)$ time per back-step. In fact, it is not difficult to show that for every prefix of size $n'$ the amortized time per back-step is $O(\lg n')$.

## 3. The list pebbling algorithm

In this section we describe the list pebbling algorithm, which supports an arbitrary sequence of forward and back steps. Each forward step takes $O(1)$ time, where each back step takes $O(\lg n)$ amortized time, using $O(\lg n)$ pebbles. We will first present in Section 3.1 the basic algorithm which uses $O(\lg^2 n)$ pebbles, then describe in Section 3.2 the pebbling algorithm which uses $O(\lg n)$ pebbles.

The list pebbling algorithm is an extension of the algorithm described in Section 2.3. It uses a new set of pebbles, denoted as *green pebbles*. The pebbles used as described in Section 2 are now called *blue pebbles*. The purpose of the green pebbles is to be kept as placeholders behind the blue pebbles, as those are moved to new nodes in forward traversal. Thus, getting back into a position for which a green pebble is still in place takes $O(1)$ time.

### *3.1. The basic list-pebbling algorithm*

Define a *left subpath (right subpath)* as a path consisting of nodes that are all left children (right children). Consider the (blue-pebbled) path $p$ from the root to node $u$; that is, $u$ is the current position. We say that $v$ is a left child of $p$ if it has a right sibling that is in $p$ (that is, $v$ is not in $p$, it is a left child, and its parent is in $p$ but not the node $u$). Green pebbles are placed on right subpaths that begin at left children of $p$ (see Fig. 4). Since $p$ consists of at most $\lg n$ nodes, there are at most $\lg n$ green paths, and the number of green pebbles is at most $\lg^2 n$.

When moving forward, if the current position is an internal node, then $p$ is extended with a new node, and a new blue pebble is created. If the current position is a leaf, then the pebbles at the entire right subpath ending with that leaf are converted from blue to green (rather than removed). Consequently, all the green sub-paths that are connected to this right subpath are unpebbled. That is, their pebbles are released and can be used for new blue pebbles. When moving backward, green pebbles will become blue. Their left subpaths will not be pebbled — repebbling these subpaths will be done when needed.

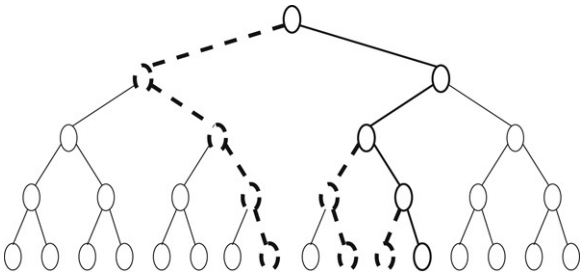

Fig. 4. Green subpaths (dashed lines) connected to the blue path (bold solid line).

We consider three types of back-steps:

(i) *Current position is a left child:* Move to the parent, which is on $p$ and is hence pebbled, and unpebble the current position.
(ii) *Current position is a right child, and a green subpath is connected to its parent:* Move to the leaf of the green subpath, convert the pebbles on this subpath to blue, and unpebble the current position.
(iii) *Current position is a right child, and a green subpath is not connected to its parent:* Reconstruct the green pebbles on the right subpath connected to its parent $v$, and act as in the second case. This reconstruction is obtained by executing forward traversal of the left subtree of $v$.

**Theorem 3.1.** *The basic list pebbling algorithm supports $O(\lg n)$ amortized list steps per back step, one list step per forward step, using $O(\lg^2 n)$ pebbles.*

**Proof.** In cases (i) and (ii) we move to pebbled nodes, which therefore takes $O(1)$ time, and no list step. In case (iii) the forward steps are through the sub-tree $T_l$ rooted at the left sibling of the current position. Note that this subtree is of the same size as the sub-tree $T_r$ rooted at the current position. Since there is no green subpath connected to its parent, it implies that at some previous point, the parent was not on the blue path. This occurs only after moving forward beyond the subtree $T_r$ of the current position, to $v'$, the right sibling of the parent of the current position. Further, it implies that since getting to $v'$ for the last time, we have not yet returned into $T_l$.

Consider the traversal between the last time that we got to $v'$, and getting back to the current position. This traversal covers the entire sub-tree $T_r$, and in particular includes all back steps starting at $v'$ and ending at the current position. We amortize the cost of forward steps within $T_l$ against the sequence of back steps within $T_r$. Since $|T_l| = |T_r|$, each forward step is amortized against one back step.

It remains to show that each back step can be charged at most $\lg n$ times. Indeed, a back step is charged only if it is within a right subtree, and for such subtree it is only charged once for each minimal traversal that covers the entire subtree. Since a back step can only be included in at most $\lg n$ different (right) subtrees, it can only be charged at most $\lg n$ times. ∎

### *3.2. The list pebbling algorithm with $O(\lg n)$ pebbles*

The basic list pebbling algorithm is improved by reducing the number of green pebbles on most of the green paths. Let $v$ be a left child of $p$ and let $v'$ be the right sibling of $v$. Call $v$ the *last left child* of $p$ if the left subpath starting at $v'$ ends at the current position; let the right subpath starting at the last left child be the *last right subpath*. Then, if $v$ is not the last left child of $p$, the number of pebbled nodes in the right subpath starting at $v$ is at all times at most the length of the left subpath in $p$, starting at $v'$ (see Fig. 5). If $v$ is the last left child of $p$, the entire right subpath starting at $v$ can be pebbled. We denote the (green) right subpath starting at $v$ as the *mirror subpath* of the (blue) left subpath starting at $v'$. Nodes in the mirror subpath and the corresponding left subpath are said to be *mirrored* according to their order in the subpaths.

**Claim 3.2.** *The number of green pebbles is at most* $\lg n$.

**Proof.** For each green subpath, except perhaps for the last one, there is a mirrored blue subpath of the same length, which is mirrored only to this green subpath. The sum of lengths of all those green subpaths is at most the length of the blue path. The length of the last green subpath is at most the distance from the current position to the leaf, which complements the length of the blue subpath to $\lg n$. ∎

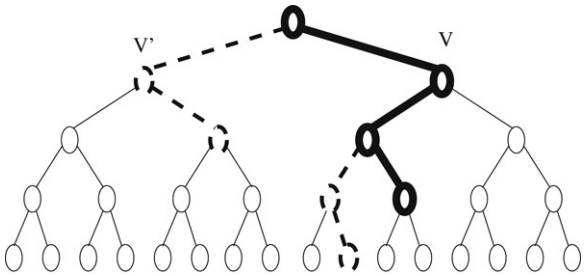


Fig. 5. A green path starting at $v'$ and its mirrored blue left subpath starting at $v$; the last green subpath is pebbled all the way to the leaf.

A sequence of forward steps and the corresponding blue path and green paths of each position is depicted in Fig. 9 When moving forward, there are two cases:

(i) *Current position is an internal node:* as before, $p$ is extended with a new node, and a new blue pebble is created. No change occurs with the green pebbles (the mirror subpath begins at the last left child of $p$).
(ii) *Current position u is a leaf that is on a right subpath starting at v (which could be u, if u is a left child):* we pebble (blue) the new position, which is the right sibling of $v$, and the pebbles at the entire right subpath ending at $u$ are converted from blue to green. Consequently, all the green subpaths that are connected to the right subpath starting at $v$ are unpebbled; and the left subpath in $p$ which ended at $v$ now ends at the parent of $v$, so the mirror (green) node to $v$ should now be unpebbled. The released pebbles can be reused for new blue pebbles.

Moving backward is similar to the basic algorithm. There are three types of back steps:

(i) *Current position is a left child:* predecessor is the parent, which is on $p$, and hence pebbled. The current position is to be unpebbled. No change occurs with green pebbles, since the last right subpath is unchanged.
(ii) *Current position u is a right child, and the (green) subpath connected to its parent is entirely pebbled:* Move to the leaf of the green subpath, convert the pebbles on this subpath to blue, and unpebble the current position. Note that the blue subpath ending at the left sibling of $u$ is now longer than its mirrored green subpath. We do not extend this green subpath, but rather defer it to when we actually need it, addressed next.
(iii) *Current position is a right child, and the (green) subpath connected to its parent is only partially pebbled:* Reconstruct the green pebbles on the right subpath connected to its parent $v$, and act as in the second case. This reconstruction is obtained by executing forward traversal of the subtree $T_l$ starting at $v$, where $v$ is the last pebbled node on the last right subpath (which is the partially pebbled green subpath).

**Theorem 3.3.** *The list pebbling algorithm supports full traversal in at most* $2\lg n$ *amortized list steps per back step, one list step per forward step, using* $2\lg n$ *pebbles.*

**Proof.** In cases (i) and (ii) we move to pebbled nodes, which therefore takes $O(1)$ time, and no list step.

In case (iii) the forward steps to reconstruct the green path are through the sub-tree $T_l$ rooted at $v$, the last pebbled node on the last right subpath. Let $v'$ be the mirrored node to $v$. Since the right child of $v$ is not pebbled, it implies that at some previous point in time, the left child of $v'$ was not on the blue path. Further, it implies that since getting to $v'$ for the last time, we have not yet returned into $T_l$.

We amortize the cost of traversing through $T_l$, which is $|T_l| - 1$ list steps, against the back traversal starting at the right child of $v'$ and ending at the current position. This back traversal includes in particular $T'_r$, which is the left subtree of $v'$. Since $|T_l| = 2|T'_r| + 1$, each back step in $T'_r$ is charged with at most two forward steps in $T_l$.

It remains to show that each back step can be charged at most $\lg n$ times. Indeed, a back step is charged only if it is within a right subtree, and for such a subtree it is only charged once for each minimal traversal that covers the entire sub-tree. Since a back step can only be included in at most $\lg n$ different (right) sub-trees, it can only be charged at most $\lg n$ times. ∎

For convenience of implementation, as well as for reducing the number of pebbles from $2\lg n$ to $\lg n$, we will in fact keep the first node in each green path unpebbled. Such nodes can be accessed in one list step from their respective parent nodes. Therefore, keeping them unpebbled adds at most one list step per back step. On the other hand, keeping them unpebbled eliminates the need to unpebble them during forward steps. Since now the node $v$ of each green

pebble is a mirrored node to a blue pebbled node $u$ which is a left child, the green pebble of $v$ will be released during a forward step that originates at $u$. Note that at this forward step, the blue pebble at $u$ is converted to green. For the sake of convenience, and based on the above, in the rest of the paper we will ignore occasionally the fact that the first node in each green path is not pebbled.

## 4. The full list pebbling algorithm

In this section we provide the full list pebbling algorithm. In Section 4.1 we refine the algorithm from Section 3 and describe an advanced algorithm that supports back steps in $O(\lg n)$ time per step in the worst case. All algorithms described so far assume that the size of the list, $n$, is known a priori; Section 4.2 shows how to obtain the same efficiency for a list of unknown size. Section 4.3 presents a "supernode" technique that enables reducing the overhead per forward steps, and that will be used also for the program rollback application of Section 6. We summarize with the LTS Theorem, the implementation of the list traversal synopsis on a RAM model.

### *4.1. The advanced list-pebbling algorithm*

Ensuring $O(\lg n)$ list steps per back step in the worst case is obtained by progressively reconstructing green paths during back traversal. The reconstruction is done using a new set of pebbles which are denoted as *red pebbles*. For each green path, there is one red pebble whose function is to progressively move forward from the deepest pebbled node in the path, to reach the next node to be pebbled. By synchronizing the progression of the red pebbles with the back-steps, we can guarantee that green paths will be appropriately pebbled whenever needed.

While a straightforward implementation with red pebbles can result with a total of $3 \lg n$ pebbles, we show how to reduce the total number of pebbles to $\lg n$. Furthermore, we show how to reduce the time per back step to be sensitive to the actual number of back-steps previously executed. Blue pebbles are saved by relying on recursive application of the list pebbling algorithm, and green pebbles are saved by delaying their creation without affecting the bound on the back-step time.

**Theorem 4.1.** *The list pebbling algorithm can be implemented using* $\lg n$ *pebbles, supporting every forward step in one list step, and every back step in* $O(\lg i)$ *list steps in the worst case, where* $i$ *is the distance from the current position to the farthest point traversed so far.*

To prove the theorem, we first implement back steps in $\lg n$ list-steps per back step, using $2 \lg n$ pebbles (Section 4.1.1). We then show how to implement forward steps in one-list step and $O(1)$ time per step, and the necessary modifications to back step implementation, so as to support arbitrary traversals (Section 4.1.2). Subsequently, we show how to reduce the number of pebbles to $\lg n$ (Section 4.1.3). Finally, we show how to reduce the back-step time to $O(\lg i)$ (Section 4.1.4).

As in the basic list pebbling algorithm we will use the blue pebbles placed on the path from the root to current position as well as an additional set of at most $\lg n$ green pebbles that will be placed in advance and be converted to blue pebbles at the appropriate time. Additionally, we will use a new set of at most $\lg n$ *red pebbles*, whose purpose is to assist in reconstructing the green paths, by traversing the appropriate sub-trees and reaching positions that were previously green pebbled.

For illustration purpose, the dynamics of the pebbling looks as follows: At all time, there are up to $\lg n$ red pebbles – one per green path – that each advance one step per back-step; their exact number equals the number of right sub-trees that include the current position. When a red pebble reaches the right child of a green pebble, this node is pebbled with a green pebble, extending the green path by one. As before, each green path is connected to a blue-pebbled node. If this node is an unpebble due to a forward step, then the green path is released. When moving backwards to the end of a green path, it becomes blue.

#### *4.1.1. Back step in at most* $\lg n$ *list steps, using* $2 \lg n$ *pebbles*

When a node $u$ is a right child, then there should be a green path from the left sibling of $u$ to node $u - 1$ (which is the rightmost leaf in the tree rooted at that sibling). We start placing the green pebbles on this green path well in advance, according to the following strategy. Let $T$ be the sub-tree of node $u$, and let $T'$ be the sub-tree of its left sibling. As we enter via a back step into $T$ (that is, moving into the rightmost leaf of $T$), and start back traversing

within $T$, we also start forward traversing the sub-tree $T'$ at the same rate, using a red pebble. Whenever the red pebble reaches a node in $T'$ that is on the (right) path from the root of $T'$ to $u-1$, we place there a green pebble. When the back traversal in $T$ reaches node $u$, the forward traversal in $T'$ ends at node $u-1$, and we have all the green pebbles of $T'$ in place. In the next back step, moving from node $u$ to node $u-1$, the green pebbles in $T'$ will be transformed into blue pebbles.

Since every node is in at most $\lg n$ sub-trees, we have at most $\lg n$ such processes occurring in parallel, and they can be implemented in a dovetailing fashion in $\lg n$ steps. Thus, the process consists of using up to $\lg n$ red pebbles, and traversing each of them one step forward for every back step, resulting with an overhead of at most $\lg n$ list steps per back step.

For each green path that ends at an internal node, there is one red pebble whose function is to assist in extending the green path. Therefore, based on Claim 3.2, the number of red pebbles is at most $\lg n$, and the total number of pebbles (including blue, green and red) is at most $3\lg n$. The total number of pebbles can be reduced to $2\lg n$ by keeping the first node of each green path unpebbled. Since this node is a left child of a blue pebble, it can always be reached in one list-step.

#### *4.1.2. Forward step in one list step and $O(1)$ time*

Moving forward involves a single list step, in which a new blue pebble is created. While moving forward, the list traversal synopsis should be maintained so as to support future back steps. This can be done with no additional list steps and $O(1)$ time, as follows:

The implementation of a forward traversal can be thought of as virtually rolling back a back-traversal implementation. By doing that, the synopsis will be ready to support a back step as needed. There are two major issues concerning a straightforward implementation of this approach. First, red pebbles cannot be moved backwards in constant time, since this would be against the direction of the list. Second, even if a red pebble could be moved backwards in constant time, the cost of forward step would be the same as that of a back step, in contrast to our requirement of $O(1)$ time per forward step.

The function of a red pebble is to extend its green path during back traversal. Thus, the converse functionality of the red pebble during forward traversal would be to shrink its green path. We note that shrinking the green path by one can be done more quickly than growing it — by unpebbling the red pebbled node, and changing the colour of the last green pebble to red. Thus, rather than having red pebbles tracing back their steps as executed during back traversal, the green paths are shrunken when needed, in $O(1)$ time. To implement this approach, we only need to know when a green path is to be shrunken — this is exactly when the mirrored blue pebble of the last green-pebbled node changes its colour to green during a forward step. Since at most one green path needs to be shrunken at any particular forward step, the number of operations on red pebbles required per forward step is $O(1)$.

Note that during forward steps red pebbles mostly remain in place. Thus, the positions of red pebbles may be different than what would be expected if we were to implement back traversal as described above; namely, advancing the red pebbles at every back step. A red pebble in its expected position is said to be *in sync*, whereas a red pebble that is in a different position is said to be *out of sync*. Note that a pebble that is out of sync could only be in a more advanced position than if it were in sync.

To keep control over the advancement of red pebbles during back traversal, we require that a pebble is advanced only if it is in sync. To support it, for every red pebble we keep a record of the current position at which it was created at its present node; this position is denoted as its *originating position*. Thus, during a back step, a red pebble is identified to be in sync exactly if its originating position is the same as the current position.

The particular changes that occur for each type of pebble is as follows:

**Blue pebbles**:

*Creation:* A new blue pebble is created in the new current position.

*Release:* When the current position $u$ is a leaf, the first (blue) pebble of the right path ending at $u$ is released, all the other blue pebbles on the right path ending at $u$, not including $u$, become green, and $u$'s pebble becomes red.

**Green pebbles**:

*Creation:* A green path may result from a colour change of a blue right path.

*Release:* A single green pebble is supposed to be released when its mirrored blue pebble is released. Rather than releasing it, its colour is changed to red.

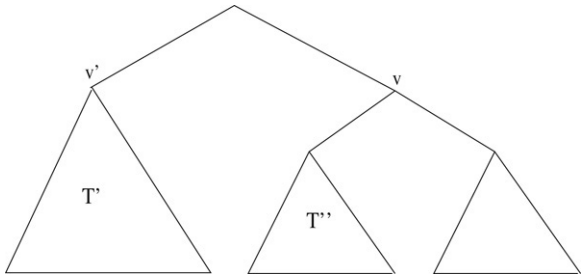


Fig. 6. In a compressed synopsis, a red pebble is created at $v'$ only when reaching $T''$.

**Red pebbles**:
*Creation:* A blue pebble at a leaf node (which is the current position) changes its colour to red upon a forward step. A green pebble which is not the first in its green path, whose mirrored blue pebble changes its colour to green (this green pebble is the last green pebble on its green path), changes its colour to red. The current position before the forward step is recorded as the originating position for the red pebble.
*Release:* A red pebble whose green path is connected to a blue node $v$ should be released when $v$ is no longer blue pebbled. This occurs during a forward step from a leaf, where $v$ is on a right blue path $p_r$ ending at that leaf. (Note that before that step, there is no green pebble left on the green path connected to $v$.) However, there are $|p_r| - 2$ red pebbles that should be released during such forward step. We limit the number of released pebbles to one per forward step, by delaying the releases of the red pebbles connected to $p_r$. These releases are scheduled to the next $|p_r| - 2$ forward steps, one per step. These forward steps will all be from a node to its left child ("Case 1") along the left path that is mirrored to $p_r$. Note that during such forward steps no other maintenance operations are required.
**Analysis.** Since a change of colour and a release of a pebble do not involve any list steps, there is indeed only one list step per forward step.

The algorithm makes no use of the information of whether a pebble is blue or green. Therefore, there is no need to actually distinguish between blue pebbles and green pebbles, and in particular no need to actually change the colour of pebbles from blue to green or vice versa — the change of colours only serves for analysis purposes. A release of a pebble takes $O(1)$ time. The change of position of red pebbles takes $O(1)$ time, and is triggered by the event causing a release of a green pebble. For each forward step, only one green or red pebble may need to be released. Therefore, each forward step can be executed in $O(1)$ time.

*4.1.3. Using* $\lg n$ *pebbles*

To reduce the total number of pebbles from $2 \lg n$ to $\lg n$, we modify the list-traversal synopsis to a *compressed synopsis* as follows:

(1) Keep all nodes on left subpaths of the blue path unpebbled; each unpebbled node on a left subpath can be reached from the parent of the first node of that subpath.
(2) Have each green path shorter by one. This is obtained by imposing a delay in the creation of the red pebbles while moving backwards. Specifically, a red pebble will be created at a root $v'$ of a tree $T'$ only when reaching the sub-tree $T''$ of the left child of the right sibling $v$ of $v'$, if such right sibling exists (see Fig. 6). To maintain a green path shorter, the release of a green pebble while moving forward occurs when the pebble at the left child of its mirrored node (rather than the node itself) change colour from blue to green. Once created, a red pebble is advanced in double pace — two list steps at a time per back step.

**Lemma 4.2.** *Using the compressed synopsis, the total number of pebbles is at most* $\lg n$*, and the number of list steps per back step is at most* $\lg n$.

**Proof.** When a red pebble is created at node $v'$, it needs to traverse the tree $T'$ rooted at $v'$ during the back traversal of $T''$, whose size is half the size of $T'$. By having the red pebble advance in double pace it is guaranteed that when the current position becomes $v$, the right path from $v'$ (the root of $T'$) to the rightmost leaf of $T'$ (which is $v - 1$), is already pebbled by green pebbles, as required.

To see that there are at most $\lg n$ pebbles, we note that in the compressed synopsis, the number of pebbles in each green subpath is smaller by one than the length of its mirrored blue left subpath. It is easy to verify that this remains

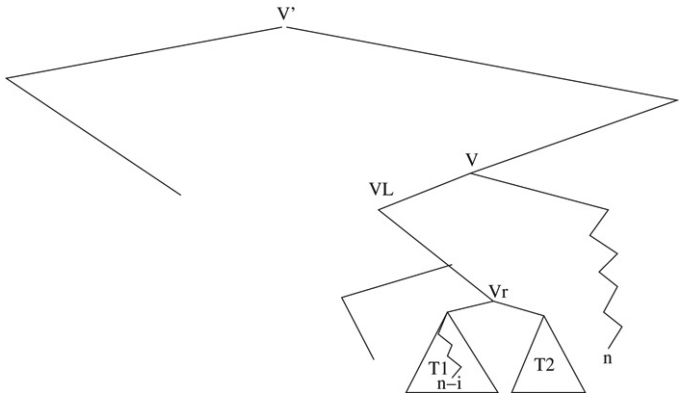


Fig. 7. The setting of the $i$-sensitive synopsis, with $\lg\ i + 1$ active red pebbles.

true despite the double pace at which the red pebbles are advanced. Since left blue subpaths are not pebbled, and there is at most one additional red pebble for each green path, the total number of pebbles is $\lg n$.

It remains to show that the number of list steps per back step is at most $\lg n$. Note that a green path is connected to a blue node which has a blue right child, which has a blue left child. Therefore, when the current position is on a left subpath of length $h$, then the number of green paths, and hence of red pebbles, could be at most $(\lg n - h)/2$. Having a back step from such current position involves at most $h$ steps on the (unpebbled) blue left subpath, and the advancement of at most $(\lg n - h)/2$ red pebbles, two list steps each, totalling $\lg n$ list steps. ∎

*4.1.4. Back step in $O(\lg i)$ list steps*

It is desirable that when executing a constant number of back steps, their cost would be constant. More generally, we would like the cost of a back step to be a function of the number of back steps actually executed. We say that the synopsis is *$i$-sensitive* if the number of list steps executed per back steps is $O(\lg i)$, where $i$ is the distance from the current position to the farthest point traversed so far.

We show that the synopsis with the modifications on the advancement of red pebbles as described in the forward step (Section 4.1.2) is $i$-sensitive. Let $v$ be the least common ancestor of nodes $n$ and $n - i$; let $v'$ be the closest ancestor of $v$ which has a connected green path. See Fig. 7. Only red pebbles that are in the sub-tree rooted by $v'$ are advanced. Indeed, during any traversal between $n - i$ and $n$, the blue path above $v'$ is unchanged. Therefore, during such traversal there is no change in any of the green paths connected to any ancestor of $v'$. Recall that red pebbles are moved backwards only during changes in green pebbles. Therefore, a red pebble that is not in the subtree rooted at $v'$ does not change position backwards during forward traversal from $n - i$ to $n$, and consequently does not need to advance forward during a back traversal from $n$ to $n - i$.

**Lemma 4.3.** *The number of red pebbles in the tree rooted at $v'$ is at most* $\lg i + 1$.

**Proof.** Let $v_L$ be the left child of $v$, and let $v_r$ be the last node on the right blue subpath starting at $v_L$. Let $T_1$ be the sub-tree rooted at the left child of $v_r$. (Note that $n - i$, $v_r$, $v_L$, $v$, and $v'$ are all on the blue path, that $n - i$ is in $T_1$, that $v_r$ may coincide with $v_L$, and that $n - i$ may coincide with $v_r$.) We observe the following:

(1) In $T_1$ there are at most $\lg i - 1$ red pebbles: Let $T_2$ be the tree rooted at the right child of $v_r$. Since all the nodes of $T_2$ are between node $n - i$ and $n$, its size is at most $i$. Hence, the size of $T_1$ is at most $i$ and its height is at most $\lg i$, implying at most $\lg i - 1$ red pebbles.
(2) There is one red pebble at the tree rooted at the left sibling of $v_r$, unless $v_r = v_L$, in which case there is no such left sibling.
(3) There is no other red pebble under $v$, since there is no left blue subpath between $v_L$ and $v_r$.
(4) There is one red pebble at the tree rooted at the left child of $v'$.

Since the above accounts for all possible red pebbles in the sub-tree rooted at $v'$, the lemma follows. ∎

Note that the "farthest point reached so far" can be reset at any time to be considered node $v$, by advancing the red pebbles to the positions that correspond to $v$.

**Using $\lg n$ pebbles.** The above implementation of the $i$-sensitive synopsis assumes $2\lg n$ pebbles. Using a compressed synopsis (Section 4.1.3) would reduce the number of pebbles to at most $\lg n$, as desired. However, the compressed synopsis would not enable $O(\lg i)$ per back step, since reaching unpebbled nodes on a left blue sub-path of length $k$ requires up to $k-1$ steps, and $k$ can be as large as $\lg n$. To obtain simultaneously at most $\lg n$ pebbles and $O(\lg i)$ time per back step, we extend the compressed synopsis to an *enhanced compressed synopsis*, in which a recursive data structure is maintained over the blue left subpaths, as follows:

Each left subpath is considered as a linked list, and we implement backward steps along a left subpath by using the above algorithm recursively. For such implementation, a left subpath of length $k$ requires $\lg k$ pebbles. We show that such pebbles are available without increasing the total number of pebbles, by observing that the mirrored right path requires only $k - \lg k$, rather than $k$, green pebbles. Thus, we can use $\lg k$ pebbles for the recursive data structure of the left subpath.

**Claim 4.4.** *Suppose that the current position $u$ is on a left subpath $p$ starting at $v$, and that the distance between $v$ and $u$ is $k$ (i.e. $u = v + k$). Then, the number of green pebbles on the mirrored right subpath starting at $v'$, the left sibling of $v$, is $k - \lg k - 1$.*

**Proof.** Recall that the green pebbles on the right subpath starting at $v'$ are constructed during a back traversal on a tree $T''$ rooted at the left child of $v$ — two forward steps are executed in the tree $T'$ rooted at $v'$ for each back step within $T''$. Since $u$ is $k$ steps far from completing the back traversal in $T''$, there is a sub-tree $T'''$ of size $2k$ within $T'$ that is yet to be traversed before constructing its green pebbles. The number of green pebbles within $T'''$ is $\lg k + 1$. ∎

### *4.2. Growing and shrinking the virtual tree for an unbounded sequence*

Suppose that we have a list traversal synopsis on a virtual pre-order tree $T$ of $n$ nodes. If the current position is $n$, then the blue path consists of the right path of $T$. When moving forward from node $n$ to node $n+1$, the tree $T$ can no longer be used for the list traversal synopsis. Instead, a (virtual) tree $T'$ of size $2n+1$ is to be used.

Our objective is to re-pebble the nodes so as to obtain the blue path of $n+1$ in $T'$. This blue path consists of the root, followed by the right path of the tree $T''$ — the left sub-tree of $T'$. Note that $T''$ is of the same size of $T$ and therefore there is a natural correspondence between nodes in $T''$ to nodes in $T$. Further, note that the pre-order numbering of each node in $T''$ is one more than the pre-order numbering of its corresponding node in $T$, since the root of $T''$ has a pre-order numbering two. Thus, the blue path for node $n+1$ in $T'$ is obtained by moving each blue pebble (from the blue path of $n$) to its next position, and adding a new blue pebble to the root of $T'$.

The amortized cost of computing the new blue path is $\lg n/n$, and such update can be prepared in advance in a straightforward manner, so as to keep the time per forward step $O(1)$ in the worst case. The above implies executing some list-steps during forward steps. This can be avoided using the following observation: each of the pebbles which are behind will either be released or otherwise will be required to support back-step. If it is to be released then there is no need to fix its position. If it is required to support a back step, it means that the current position is within the tree $T$, and therefore we can use again $T$ as the virtual pre-order tree in which case the pebble is in its correct position.

**Reducing the number of pebbles during back traversal.** The number of pebbles can also be kept at all time $O(\lg n)$, where $n$ is the current position, even if at some point in the past the virtual tree grew to arbitrary size $N \gg n$. Indeed, the blue path consists of a left path of some length $n_1$, and some blue path within a subtree $S$ of size at most $2n_2$, so that $n_1 + n_2 = n$. As in the advanced algorithm using $\lg n$ pebbles, the left blue path can be substituted with a recursive instance of a list traversal synopsis of size $O(\lg n_1)$. Since the list traversal synopsis of the subtree $S$ is of size $O(\lg n_2)$, we have a total of $O(\lg n)$ pebbles.

### *4.3. Supernodes for $\epsilon$ overhead in forward steps*

The overhead in forward step was shown to be $O(1)$ in the worst case. We show how to reduce the amortized time overhead per forward step to $\epsilon$, for arbitrary small $\epsilon > 0$. Let $c > 0$ be a constant so that each forward step takes at most $c$ time. Given a list $L$ of length $n$, we define a virtual list $L'$ over $L$, as follows: each node in $L'$ is a supernode, representing a group of $c/\epsilon''$ consecutive nodes in $L$, where $\epsilon'' < \epsilon$ will be defined below. We keep a list traversal synopsis over $L'$, so that whenever a pebble is allocated to a supernode in $L'$, we will instead allocate the pebble to the first node of $L$ within that supernode.

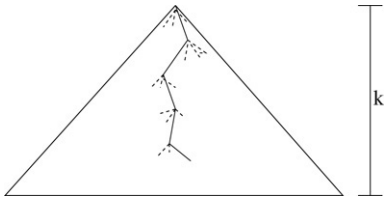


Fig. 8. The $n^{1/k}$-ary virtual pre-order tree, supporting $kn^{1/k}$ vs. $k$ trade-off.

A forward step in $L'$ from a supernode $v'$ to a subsequent supernode $u'$ occurs only when there is a forward step in $L$ from the last node in $v'$ to the first node in $u'$. Since the time per forward step in $L'$ is at most $c$, the amortized time per forward step in $L$ is at most $\epsilon' + c/(c/\epsilon'') = \epsilon' + \epsilon'' = \epsilon$ where $\epsilon'$ is the overhead per step due to managing the supernodes, and $\epsilon''$ defined as $\epsilon - \epsilon'$. Note that $\epsilon'$ will include the overhead of counting list steps within a supernode. Such counting could be avoided by using a skipping command such as do_i_forward_step(i), or if a supernode is not defined by the number of list steps, but is defined instead by a time-driven system interrupt. This enables the overhead per supernode to be constant — independent of the supernode size. As a result, $\epsilon'$, and hence $\epsilon$, can be made arbitrarily small.

A backward step in $L$ can be of two types. If the back-step is between two nodes belonging to the same super-node, then it is executed by moving forward at most $c/\epsilon'' = (1/\epsilon)$ steps from the beginning of the shared supernode; no back step occurs in $L'$ in this case. If the back step is between two nodes belonging to a different supernodes, then a back step in $L'$ is executed, and the required node is reached by moving forward $O(1/\epsilon)$ steps from the beginning of its supernode. Executing a back step in $L'$ involves a sequence of forward list steps in $L'$; each such list step is executed by having $c/\epsilon''$ forward steps in $L'$. However, in general each back step in $L'$ occurs after $c/\epsilon''$ back steps in $L$. Hence, the time per back step is now increased by an additive factor of $O(1/\epsilon)$.

We can summarize with the following theorem, regarding the implementation of the list traversal synopsis (LTS) on a random access machine model:

**Theorem 4.5** (*The LTS Theorem*). *The list pebbling algorithm can be implemented on a RAM using* $\lg n$ *pebbles and* $O(\lg n)$ *memory, where* $n$ *is the distance from the beginning of the list. It supports* $O(\min\{\lg n, \lg i\})$ *time in the worst case per back step, where* $i$ *is the distance from the current position to the farthest point traversed so far. Each forward step takes* $O(1)$ *time and one list step in the worst case. The amortized time overhead can be at most* $\epsilon$ *per forward step, for arbitrary* $\epsilon > 0$*, by increasing the time per back step with an additive overhead of* $O(1/\epsilon)$.

## 5. Obtaining a full time-memory trade-off

In order to support a full time-memory trade-off, we extend the virtual pre-order tree data structure from a binary tree into a $n^{1/k}$-ary tree (see Fig. 8). The same virtual $n^{1/k}$-ary tree serves both the case of having $O(k)$ pebbles with $O(kn^{1/k})$ time, and the case of having $O(kn^{1/k})$ pebbles and $O(k)$ time, but with different placements of pebbles. It consists of a virtual pre-order tree data structure of depth $k$ and degree $n^{1/k}$. As for the binary tree, in both cases we maintain a blue pebble path from the root to the current position (of length at most $k$). The difference is in the placement of the green pebbles.

For the case of $O(k)$ pebbles and $O(kn^{1/k})$ time per back step, we have the green paths begin only at the nearest left sibling of each node with a blue pebble. As in the binary case, they consist of right sub-paths and their lengths are according to the mirroring property. As in the binary case, the total number of green pebbles is at most the total number of blue pebbles, which is at most $k$. Hence, we have at most $2k$ pebbles in the tree. The time per back-step is dominated by the time it takes to reconstruct a new green path. Consider a back step from a root $v$ of a sub-tree $T$. Constructing the green path that starts at the parent of $v$ involves forward traversal through at most $n^{1/k}$ sub-trees identical to $T$. As in the binary case, the construction time is amortized against the sequence of back steps from the right-most leaf of $T$ till the root of $T$. Since each node is in at most $k$ different trees, and hence each back step is amortized against at most $k$ such green-path constructions, the amortized time per back step is $O(kn^{1/k})$. Red pebbles can be used as in the binary case to enable worst-case performance, increasing the total number of pebbles to at most $3k$. Similar techniques to the binary case can be used to reduce the number of pebbles. Forward steps take $O(1)$ each as before.

For the full trade-off in which the number of pebbles is $O(kn^{1/k})$ and the time per back step is $O(k)$, we use the same virtual tree of depth $k$. We place blue and green pebbles as before, but in addition we place green pebbles at all left siblings of each node with a blue or green pebble. The number of pebbles is at most $n^{1/k}$ times the number of pebbles in the previous case, that is at most $2kn^{1/k}$. The amortized time per back step here is $O(k)$. Indeed, the same amortization argument of the previous case applies here, except that constructions of green paths only involve forward traversal of one subtree. Worst-case performance is again obtained using red pebbles, and $O(1)$ time per forward step as well as reducing the number of pebbles is obtained as before.

## 6. Reversal of program execution

A unidirectional linked list can represent the execution of programs. Program states can be represented as nodes in a list, and a program step, or a sequence of steps, is represented by a directed link between the nodes representing the appropriate program states. Since typically program states cannot be easily reversed, the list is in general unidirectional.

Consider a linked list that represents a particular program execution. Moving from a node in the list back to its preceding node is equivalent to reversing the step represented by the link. Executing a back traversal on the linked list is hence equivalent to rolling back the program. Let the sequence of program states in a forward execution be $s_0, s_1, \ldots, s_T$. A *rollback* of a program of some state $s_j$ is changing its state to the preceding state $s_{j-1}$. A rollback step from state $s_j$ is said to be the *i'th rollback step* if state $s_{j+i-1}$ is the farthest state that the program has reached so far.

First, we show how to efficiently support back traversal with negligible overhead to forward steps in programs for which the program states can be represented using a small amount of memory, $S$. Note that $S$ may be considerably smaller than the memory used by the program, and the size required to encode the entire configuration state of the program. An application for such case is described below in Section 6.2; additional applications are described in [7]. For simplicity, we assume that two consecutive program states are separated by a sequence of $\ell$ program steps. The following theorem follows directly from Theorem 4.5, by using the supernode approach with $\ell$ nodes in each supernode, and noting that a list step which leaves a state copy, involves copying the state information in $O(S)$ time.

**Theorem 6.1.** *Let $P$ be a program, whose state information can be represented using memory of size $S$, and whose running time is at most $T$. Then, at the cost of recording the input used by the program, and increasing the memory by an additive factor of $O(S \lg(T/\ell))$, the program can be extended to support arbitrary rollback steps as follows. The i'th rollback step takes $O(S \lg i + \ell)$ time in the worst case, while forward steps take $O(S)$ time in the worst case, and $O(S/\ell)$ amortized time per step.*

Next, we show how to rollback efficiently an arbitrary program, for which the state information can be as large as the entire memory used by the program.

### 6.1. Program rollback with delta encoding

Program rollback techniques have important applications, including debuggers and backward simulations. Traditional methods of supporting rollback are based on check-pointing particular program states. However, it is customary to also utilize the fact that the change in a program state during a single program step is often substantially smaller than the size of the program state. Indeed, rather than just recording program states, one can record the differences needed in program states in order to convert them into their preceding states, often called *delta-encoding*. This results with better memory utilization, at the cost of additional overhead per forward step.

A transition from a program state to the next state, either in a single step or through a sequence of steps, may involve several instructions. A transition that takes $t$ instructions generates a delta of size $\Delta = O(t)$. Hence, the time to record the delta at each step is proportional to the time of the step itself. For simplicity, we assume that all program steps take the same time $t$. Suppose that the delta-encoding is smaller than the program state by a factor of $\ell$; that is, $\ell \approx S/t$. Then, after $\ell$ steps the accumulated size of delta-encoding is about the size of a single program state. The rollback method of Theorem 6.1 can be enhanced with the delta-encoding method, by utilizing an extension of the supernode technique described above. Each supernode consists of $\ell$ nodes in the original list. Additionally, the last $\ell$ steps are fully kept using delta-encoding, adding at most the size of a single program state.

Forward steps are implemented as before, except that the oldest step kept in delta-encoding is removed, and the delta-encoding of the new step is added. The number of forward steps implemented to support the $i$'th back-step is $O(\ell \lg(i/\ell)/\ell)$, which is $O(\lg(i/\ell))$ in the worst case. By delaying the maintenance of list traversal synopsis, each of the first $\ell$ back steps can be implemented in $O(1)$ time, using the available delta-encoding. Still, for any $i > \ell$, the sequence of $i$ back steps will take $O(i \lg(i/\ell))$ in the worst case. The theorem below assumes that the information of the program state is $S = O(M)$, where $M$ is the memory size of the program. Thus, it makes no assumption about state encoding, and applies to any program. We denote the cost of a step to be the overhead incurred over the cost of a single forward step.

**Theorem 6.2** (*Rollback Theorem*). *Let $P$ be a program, using memory of size $M$ and time $T$. Let $\Delta$ be the memory size of the delta-encoding in a single step, and $\ell$ be greater than $M/\Delta$. Then, at the cost of recording the input used by the program, and increasing the memory by a factor of $O(\lg(T/\ell))$ to $O(M \lg(T/\ell) + \ell\Delta)$, the program can be extended to support arbitrary rollback steps as follows. The cost of the $i$'th back step is $O(\lg(i/\ell))$ in the worst case, while the time cost of each forward step is $O(1)$ in the worst case. The amortized time overhead beyond the cost of delta encoding is $\epsilon = O(1/\ell)$ per forward step.*

### 6.2. Hash chains

Let $h$ be a one-way cryptographic function. A hash chain for $h$ is a sequence of hash values $v_0, v_1, \ldots, v_n$, obtained by repeatedly applying $h$, starting with a secret random seed $s$. In particular, $v_0 = s$, and for all $i > 0$, $v_i = h(v_{i-1})$.

When the seed $s$ is known, then the entire chain can be easily computed. However, for a party that knows $v_i$ but does not know any $v_j$ for $j < i$, the task of computing $v_{i-1}$ is intractable. On the other hand, having given $v_{i-1}$, it can easily verify that $v_i = h(v_{i-1})$. Hash chains are attractive in their abilities to provide a low-cost, long sequence of such verification steps, where each verification involves a back step along the list representing the hash-chain. Thus, the application of a hash chain is quite similar to a full program rollback, except that rather than keeping program states, it is sufficient to keep hash values, along with the hash function. Note that in this case, we are not concerned with forward traversals.

Our list-traversal synopsis algorithm provides efficient processing of hash chains. By keeping $k$ hash values, it enables to get each preceding hash value in $O(kn^{1/k})$ time in the worst case. By keeping $kn^{1/k}$ hash values, the time per preceding hash value is $O(k)$ in the worst case. As a particular case, by keeping $\lg n$ hash values, the time to obtain a preceding hash value is $O(\lg n)$.

The cryptographic applications of hash chain include password authentication [14], micro-payments [19], forward-secure signatures [10,13], and broadcast authentication protocol [17]. Such applications and others can benefit from efficient hash-chain processing, especially in memory-challenged platforms such as smart cards. Several recent works have concentrated on effective hash-chain back-traversal [11,5,21] and related traversals of merkle trees [12,24].

## 7. Graph traversal synopses

We discuss traversal implementations for more general linked structures, such as trees, directed acyclic graphs, and general directed graphs. A traversal on such structures may be more flexible than on a linked list for two reasons: first, after backing up to a node whose out-degree is more than one, a subsequent forward step may be to a node different from previous forward steps from the same node; second, the forward traversal may include loops. A back step from node $u$ to node $v$ *invalidates* the last forward step from node $v$ to node $u$. Forward steps that were not invalidated are considered *valid*.

At every point, we denote as the *ad-hoc list* the list obtained by taking the sequence of valid forward steps from the starting point of traversal to the current position. A traversal on a given directed graph $G$ consists of a sequence of forward steps and back steps on the ad-hoc list: a forward step is from the current position $u$ to any node $w$ such that an edge $(u, w)$ exists in $G$; a back step on the ad-hoc list is defined by moving from the current position $u$ to the preceding node $v$ on the ad-hoc list.

To support traversal on a graph $G$, we define a *graph traversal synopsis* which consists of the list traversal synopsis for the ad-hoc list. We rely on the fact that the list traversal synopsis is always defined only on the nodes between the starting node and the current position of the list. Therefore, it can be equally defined on the ad-hoc list, without any

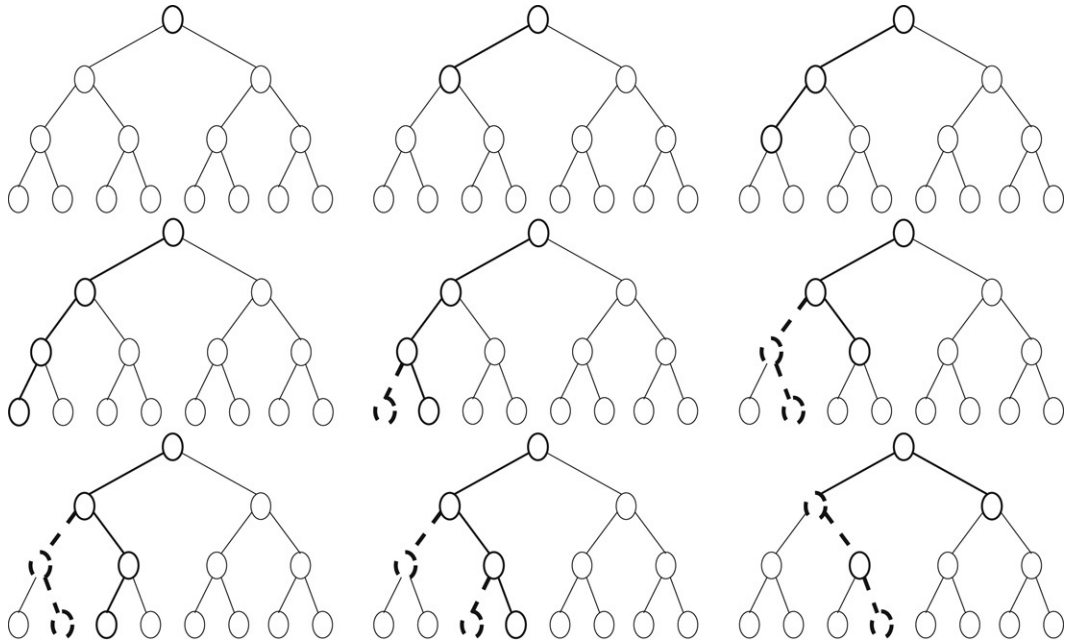

Fig. 9. The list traversal synopses in nine steps of forward traversal from the beginning of the list (left to right, top to bottom). Nodes in bold face are blue pebbled, and bold edges constitute the blue paths. Fragmented nodes are green pebbled and fragmented edges constitute green paths.

consideration to the rest of the graph. In particular, this allows to have the flexibility in forward steps, unrelated to previous forward steps from the same nodes.

The only modification that may be required to support the graph traversal synopsis is to support the *identification* of the ad-hoc list. When advancing a pebble in the ad-hoc list from a node $v$ whose out-degree $d(v) > 1$, we should know which of its neighbours $w$ in $G$ ($(v, w) \in E(G)$) is its successor on the ad-hoc list (for this particular instance of $v$ in the ad-hoc list). In general, we require to encode the ad-hoc list by identifying for each node its successor in the ad-hoc list, using $\lg d(v)$ bits. In many cases, however, such encoding would not be necessary, as the ad-hoc list will be defined by the node description, or by the application that uses the graph traversal synopsis. For instance, hierarchical structures that identify nodes by their logical path such as file pathname descriptors or a URL define the ad-hoc list without additional necessary encoding.

## 8. Conclusions

We presented efficient pebbling techniques, based on a novel virtual pre-order tree data structure, that enable compact and efficient list traversal synopses. These synopses support effective back-traversals on unidirectional lists, trees, and graphs with negligible slowdown in forward steps. In addition to straightforward applications to arbitrary traversals on unidirectional linked structures, we derive a general method for supporting efficient roll-back in arbitrary programs, with small memory overhead and virtually no effect on their forward steps. Other applications include memory- and time-efficient implementations of hash chains, with full time-space trade-off.

In a recent work, we implemented and tested the algorithms, and their theoretical bounds were shown to reflect their actual performance [7]. The implementation used the notion of the PSP model as described in the introduction, having the same implementation of list traversal synopsis used for a variety of applications. The implementation addressed various algorithm engineering considerations so as to obtain effective running time. It also incorporates an alternative technique for effectively supporting the growing and shrinking of the list traversal synopsis as the current traversal position changes. In particular, the virtual tree is extended to a *virtual forest*, which enables avoiding recursions and is simpler and more attractive for implementation. An implemented case study of the program rollback, based on Theorem 6.1, is reported in [7].

## Acknowledgments

We thank Martin Dietzfelbinger and an anonymous referee for most helpful comments. The first author's research supported in part by grants from the ISF. The second author's research supported in part by grants from the GIF.

## References

[1] A.M. Ben-Amram, H. Petersen, Backing up in singly linked lists, in: ACM Symposium on Theory of Computing, 1999, pp. 780–786.
[2] M.A. Bender, A. Fernandez, D. Ron, A. Sahai, S.P. Vadhan, The power of a pebble: Exploring and mapping directed graphs, in: ACM Symposium on Theory of Computing, 1998, pp. 269–278.
[3] A.K. Chandra, Efficient compilation of linear recursive programs, in: 14th Annual IEEE Symposium on Foundations of Computer Science, 1973, pp. 16–25.
[4] Y.C. Chung, S.-M. Moon, K. Ebcioglu, D. Sahlin, Reducing sweep time for a nearly empty heap, in: 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2000.
[5] D. Coppersmith, M. Jakobsson, Almost optimal hash sequence traversal, in: Fifth Conference on Financial Cryptography, 2002, pp. 102–119.
[6] M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows, in: Proc. of the 13th annual ACM-SIAM symposium on Discrete algorithms, 2002, pp. 635–644.
[7] M. Furman, Y. Matias, E. Porat, LTS: The List-Traversal Synopses System, in: NGITS, 2006, pp. 353–354.
[8] P. Gibbons, Y. Matias, Synopsis data structures for massive data sets, in: Proc. of the tenth annual ACM-SIAM symposium on Discrete algorithms, 1999, pp. 909–910.
[9] D.S. Hirschberg, S.S. Seiden, A bounded-space tree traversal algorithm, Information Processing Letters 47 (4) (1993) 215–219.
[10] G. Itkis, L. Reyzin, Gene itkis and leonid reyzin and verifying, in: CRYPTO, 2001, pp. 332–354.
[11] M. Jakobsson, Fractal hash sequence representation and traversal, in: IEEE International Symposium on Information Theory, 2002, p. 437.
[12] M. Jakobsson, T. Leighton, S. Micali, M. Szydlo, Fractal merkle tree representation and traversal, in: CT-RSA, 2003, pp. 314–326.
[13] A. Kozlov, L. Reyzin, Forward-secure signatures with fast key update, in: Third Conference on Security in Communication Networks, 2002, pp. 241–256.
[14] L. Lamport, Password authentification with insecure communication, Communications of the ACM 24 (11) (1981) 770–772.
[15] Y. Matias, E. Porat, Efficient pebbling for list traversal synopses, Technical Report, Tel Aviv University, 2001 (revised 2002).
[16] Y. Matias, E. Porat, Efficient pebbling for list traversal synopses, in: International Colloquium on Automata, Languages and Programming, 2003, pp. 918–928.
[17] A. Perrig, The biba one-time signature and broadcast authentication protocol, in: ACM Conference on Computer and Communications Security, 2001, pp. 28–37.
[18] N. Pippenger, Advances in pebbling, in: International Colloquium on Automata, Languages and Programming, 1982, pp. 407–417.
[19] R.L. Rivest, A. Shamir, Payword and micromint: Two simple micropayment schemes, in: Security Protocols Workshop, 1996, pp. 69–87.
[20] H. Schorr, W.M. Waite, An efficient machineindependent procedure for garbage collection in various list structures, Communications of the ACM 10 (8) (1967) 501–506.
[21] Y. Sella, On the computation-storage trade-offs of hash chain traversal, in: Financial Cryptography, 2003, pp. 270–285.
[22] L. Siklossy, Fast and read-only algorithms for traversing trees without an auxiliary stack, Information Processing Letters 1 (1972) 149–152.
[23] J. Sobel, D.P. Friedman, Recycling continuations, ACM SIGPLAN International Conference on Functional Programming 34 (1) (1999) 251–260.
[24] M. Szydlo, Merkle tree traversal in log space and time, in: Eurocrypt, 2004, pp. 541–554.
[25] D. Walker, J.G. Morrisett, Alias types for recursive data structures, in: Types in Compilation, 2000, pp. 177–206.